'Pipeline of Data Transformers' as building blocks of Data Science Workflows

Shubha Shedthikere

Abstract

Most of the workflows in data-driven model development in Big-Data Systems, involve reading messages from a data stream or a textfile or a database system, using the message to do data extraction/analysis or to train/test a model and evaluate the metrics, and writing back a new message or record into a stream like a kafka queue or into a texfile or into a count store such as Redis. These operations can be abstracted as flow of messages from a data-source into a model, which then generates a new message and writes it into a data-sink. In this write-up, we introduce a Python library, which abstracts these operations into a component, what we call, as **Transformer**, which would essentially consist of a *data-source, a model and a data-sink* and the whole workflow of training and testing the models would then boil down to transforming the messages using one transformer followed by the other, thus constituting a chain of transformers, what we call as **Pipeline**. In some sense, this is what ETL (Extract, Transform, Load) process does and this architecture is very similar to that of Apache Storm and StreamParse. This work distills these ideas into a very light-weight and highly customized library, which could prove to be very useful in building pipelines for training and testing our models.

1 Introduction

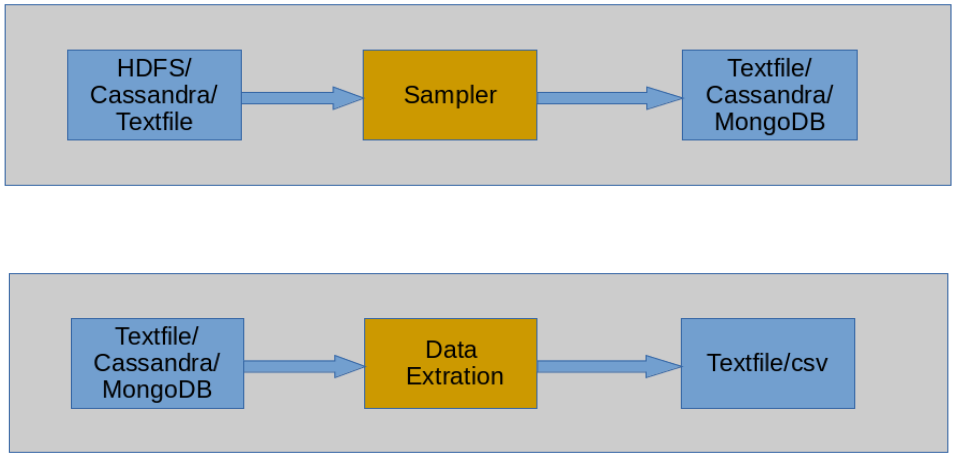
In an Agile software development model, also known as Incremental model, software is developed in incremental, rapid cycles. Each release builds on previous functionality and each release is thoroughly tested to ensure that the software quality is maintained. Unlike the more traditional waterfall model, where the requirements-freeze happens early on, an Agile model is very dynamic and needs a very lean and well designed workflows so that the new requirements can be incorporated, validated and deployed with minimum lead time.

With the advent of data-driven modelling methodologies and smart systems based on machine learning, the dataflows to draw data from various sources, to load it into database systems, to clean it, to extract features from the data and use it to build models coupled with cut-throat competition to have minimum time-to-market pose new set of challenges in the design of software development process and workflows. In such a fast-paced development environment it becomes absolutely essential for Data Science Teams to be equiped with a framework, which helps them in the workflows associated with Data-driven model development. In this work, we present one such Python library, which essentially provides a framework to interface with various data sources and has many in-built functionalities which are commonly required in the model development. We will first take a brief look at the workflows in a Data Science team and then show how this library abstracts all these workflow into a common framework.

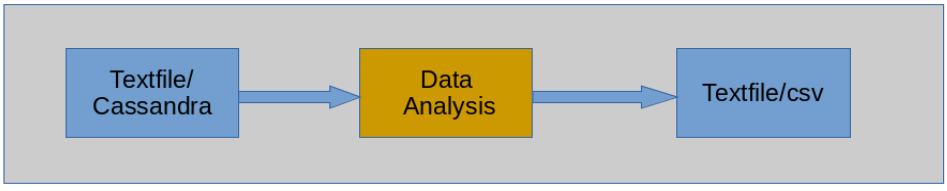
2 Data Science Workflow

Let us now review the workflows involved in a typical data science team and the associated dataflows and see how these dataflows can be abstracted so that we could build a generic framework to cater to these workflows.

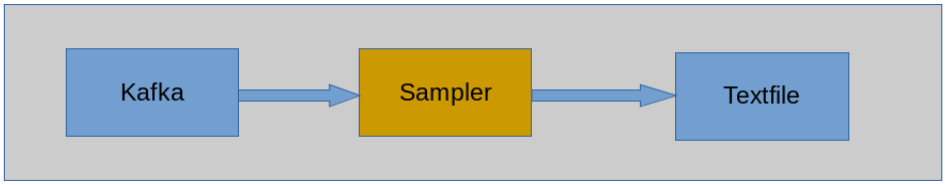
- **Data Preparation :** The first step in any data-driven modelling is to obtain the data from sources and prepare the data into a format which is amenable for data analysis. Data preparation step involves obtaining data from database systems like HDFS, Cassandra, MongoDB or streaming data from Kafka or also in the form of textfiles, cleaning up the data and extracting only that information that is relevant to the model we are trying to build. We would want to extract only a few samples of the records and use it for analysis. These operations can be represented as shown below. There is a data source, such a HDFS, a sampler, which could randomly choose some of the incoming messages and write it into a textfile, which is a data sink. These records which were written into the textfile could be cleaned up and written into another textfile, which can then be used for data analysis.



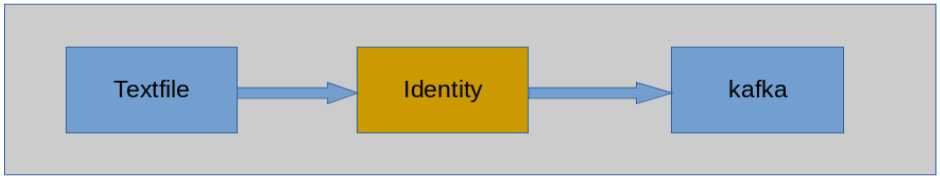
- **Data Analysis:** The data preparation step cleans up the raw data and makes it available for analysis. During the analysis stage, the prepared data could be read in, processed and the results of the analysis could be written to a file, which can then be used to obtain plots or be used by data visualization tools.

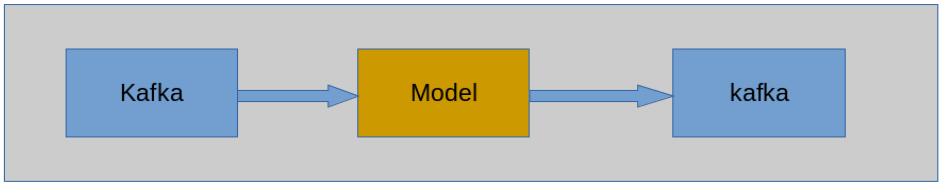


- **Model Building:** The next stage in the work flow would be modelling itself. To build the models based on machine learning we need training data and test data sets. We could use a sampler which samples from the streaming data or database system and creates the training and test data sets and write it to a file.

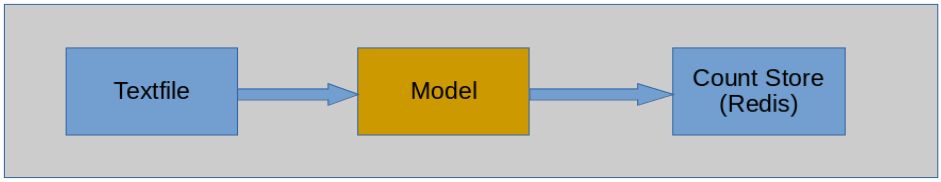


This data could be either directly used to train the model or could be read into another auxillary stream, such as a test-kafka, and then be used to train/test the models and the output of the models could be written on to another output stream, which further could be connected to count store such as Redis.

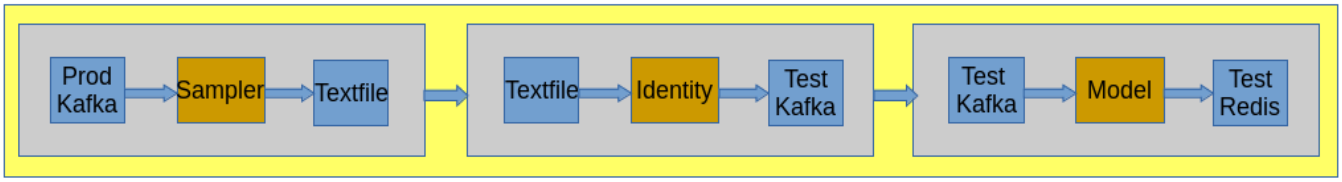




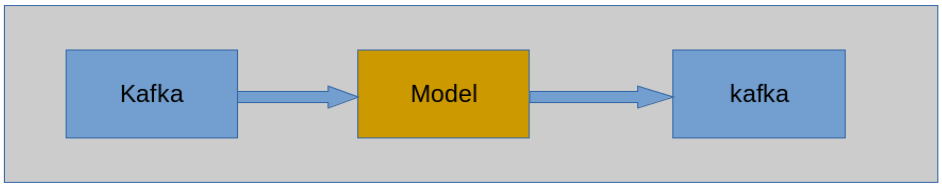
- **Functional(Static) Testing:** Once the model is built, functional testing could be performed on a new set of data. Data could be obtained as it was done during the model building stage. In this stage, we would also be interested in analysis of the output of the models, for which purpose, we could write the output of the model onto a count store like Redis and also store a metric store such as MongoDB



All these operations could be put together into a one dataflow pipeline as shown below.



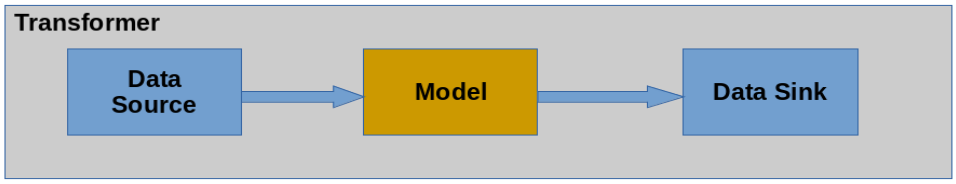
- **A/B Testing:** For A/B testing, the model could obtain messages from the same source as the model in production, run the new models and write back the new messages into a data sink such as database or output stream from where the comparison can be made between the two models.



3 Pipeline of Transformers

3.1 Transformer

From the above discussion, we see that most of the workflows require reading from a source, processing the read the information, transforming it into a new set of information and writing it to a data sink. This can be abstracted into a component, what we call as **Transformer**. A transformer consists of a data-source, model and a data-sink. The data-source and data-sink could both be any of the database systems or data streams or a textfile. Depending on the workflow, different sources, sinks and models can be instantiated based on the configuration parameters sent to the transformer.



The pseudo code below gives what the transformer would have and what operations it performs.

```
class Transformer():
    def __init__(config_paramters):
        source=DataSource(config_parameters)
        sink=DataSink(config_parameters)
        model=Model(config_parameters)

    ''' this method reads a message from source and
     sends it to the model and obtains the output
     and writes the new message onto the sink'''
    def transform_data():
        for message in source:
            output=model(message)
            sink.put(output)
```

The DataSource class basically contains the code to interface with various sources, such as textfile, Kafka queue, HDFS and give out the message in the JSON format. This layer will thus shields the model developer from the details of interfacing with different sources and different data formats.

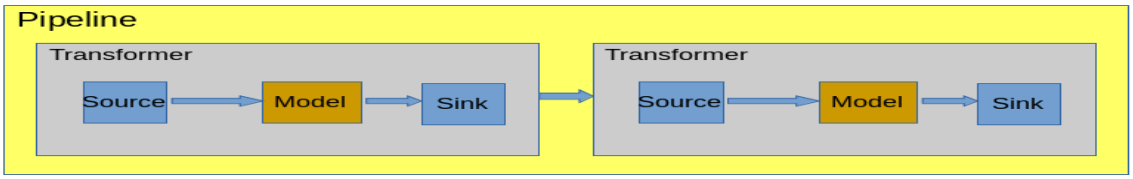
```
class DataSource():
    def __init__(self, data_source_config_parameters):
        # Initialize the source type
        self.data_source_name = data_source_config_parameters["name"]

        # Instantiate the data source components
        if data_source_config_parameters["name"] == "kafka":
            try:
                self.src_iterator = KafkaConsumer(.....)
                self.src_iterator.subscribe(....))
            except:
                print "Could not Kafka consumer.\
                    Make sure the configuration parameters are correct in
                    the pipeline builder " \

                exit()


        elif data_source_config_parameters["name"] == "file":
            try:
                self.src_iterator = open(.....)
            except:
                print "Could not open the file.\
                    Make sure the configuration parameters are correct in\
                    the pipeline builder " \


        # Please put interfaces to additional sources here.


        else:
            print "Please check the config in the pipeline builder."
```

Similarly the DataSink will contain interfaces to kafka,textfile, Redis, MongoDB. Depending on the data source (or data sink) specified in the configuration parameters of the transformer, the appropriate messages are read(or written) and given to (from) the model.

3.2 Pipeline

A workflow like training the model would require many transformers put together, where on transformer's output is input to the other transformer. Therefore pipeline would essentially consist of **transformers**, with the **connections** between the transformers specified through the configuration parameters.

Pipeline class essentially makes builds the connections between the transformers,in the sense it will take the output of one transformer and feed it to the next transformer. It has builder method, which takes a list of transformers and connection method which gives the connection between transformer.



3.3 Key Advantages:

Some of the key advantages of having such a framework would be:

- **Resuability:** Each model developer need not write his/her code to access these data-sources and sinks. Also ability to access new sources and sinks could be added and that would be available for all the users.
- Model developer need not worry about the details of the interfaces of the different sources, whether it is Redis/Kafka/HDFS, all the user